NASA Computational Mobility

Final Report

Grant No. NCC2-1413

Project Performance Period – July 1, 2003 to September 30, 2004

Institute for Human & Machine Cognition
40 S. Alcaniz St.
Pensacola, FL 32502

Final Report Summary

This blue sky study was conducted in order to study the feasibility and scope of the notion of Computational Mobility to potential NASA applications such as control of multiple robotic platforms. The study was started on July 1st, 2003 and concluded on September 30th, 2004. During the course of that period, four meetings were held for the participants to meet and discuss the concept, its viability, and potential applications. The study involved, at various stages, the following personnel: James Allen (IHMC), Alberto Canas (IHMC), Daniel Cooke (Texas Tech), Kenneth Ford (IHMC - PI), Patrick Hayes (IHMC), Butler Hine (NASA), Robert Morris (NASA), Liam Pedersen (NASA), Jerry Pratt (IHMC), Raul Saavedra (IHMC), Niranjan Suri (IHMC), and Milind Tambe (USC).

A white paper describing the notion of a Process Integrated Mechanism (PIM) was generated as a result of this study. The white paper is attached to this report. In addition, a number of presentations were generated during the four meetings, which are included in this report. Finally, an execution platform and a simulation environment were developed, which are available upon request from Niranjan Suri (nsuri@ihmc.us).

Process Integrated Mechanisms (PIMs)

1. Motivation

We are concerned with the control and coordination of teams of semi-autonomous robots engaged in complex tasks requiring coordinated action in uncertain and possibly hostile environments to achieve complex and changing goals.

There are presently no satisfactory techniques for reliably coordinating such teams in realistically complex environments. The obvious and traditional approach is to include in the team a single coordinating authority that directs and coordinates the activities of all team members. This approach, however, has difficulties. There is a high communication overhead because the coordinating authority needs to have complete and up-to-date information about the operational state of each of the robots. In addition, the overall system is inherently fragile, as any damage to the coordinating authority can render the entire team leaderless. The chief advantage of having a single coordinating authority, however, is simplicity of implementation and predictability of overall team behavior.

Agent-based approaches attempt to address the problems mentioned above. Each robot enjoys "agent-hood" and is responsible for its own actions and maintaining its own world-view. Coordination amongst the agents can require something akin to social negotiation with all its concomitant uncertainties and high computational and communication costs. Partly as a reaction to these problems, biologically-inspired approaches attempt to avoid explicit coordination altogether. Under this view, organized behavior must emerge dynamically from the individual actions of "swarms" of simple robots. What both these approaches lack is a common viewpoint or perspective on the action of the entire team considered as an integrated system, making programming and control of these systems very difficult.

We propose a novel architecture, the Process Integrated Mechanism (PIM), which has the advantages of a single coordinating authority while avoiding the structural difficulties that have traditionally led to its rejection in complex settings. We expect PIMs to improve on all previous models with regard to coordination, security, ease of software development, and robustness.

2. The Architecture of a PIM

In the PIM architecture, the components are conceived as parts of a single mechanism, even when they are physically separated and operate asynchronously. A PIM is a mechanism integrated at the *software level* rather than by physical connection. It maintains a single unified world-view, and behavior is controlled by a single coordinating process.

The idea is to retain the perspective of the single controlling authority but abandon the notion that this process must have a fixed location within the system. Instead, we propose *moving the computational state* of the coordinating process rapidly among the component parts of the PIM. The key goal here is to gain the advantages from having a single controlling process, while avoiding the problems arising in other approaches when this process is on a single processor.

The basic engineering technique is familiar from time-sharing systems and mobile code technology, and in fact can be described as "inverse time sharing." The code that implements the coordinating process (CP) is installed on all the components, and each component maintains a

current run-time state of the CP. At any given instant, only one copy of the CP, on one of the components, is actually running, where it has full access to any local data and can directly control any locally performed activity. At some point this copy of the CP is saved, and the run-time state is transmitted to the next component, where the CP immediately continues to execute. This movement of the CP state between components is rapid compared to the necessary global reaction time of the overall system, providing the illusion that the same process is running everywhere. Importantly, the CP code itself can be programmed under this simplifying assumption: the movement of the process state is invisible to it, as well as to an external observer of the system's behavior: it is handled at the operating-system level, and can be effectively ignored at all higher levels. Note that although the architecture can be described as parallel and distributed, the coordinating process itself runs serially, and interacts with any other local processes only when it is running on the same platform as that process. Thus, the entire distributed mechanism appears to the CP programmer as a single integrated platform. What would seem to be a team of communicating autonomous robots when seen from the distributed-coordination perspective is actually a single integrated mechanism that can change its distributed shape by moving its parts, but has a single locus of control and maintains a single integrated view of its world. This architecture requires an underlying technology that provides strong mobility—the movement of the current execution state between processors. Work at IHMC has shown that such models are both feasible and practical [1] [2].

This single coordinating view is a key aspect of our model, and to retain the integrity of this perspective we impose several constraints on the systems architecture. The first is that the system's own view of the world should be identified with the computational state of the coordinating process, so that an update to the CP is, automatically, an updating of the system's worldview. The second is that the updating of this state is the *only* way that local processes can exchange information with one another. The architecture assumes that all coordination between components occurs via changes to information stored in the state of the coordinating process. An information-flow view of the architecture (see Figure 1) is a collection of interacting processes where "vertical interaction" occurs within a local component and "horizontal interaction" is handled by moving the CP state.

Although the CP code runs only intermittently on any particular component, each component of

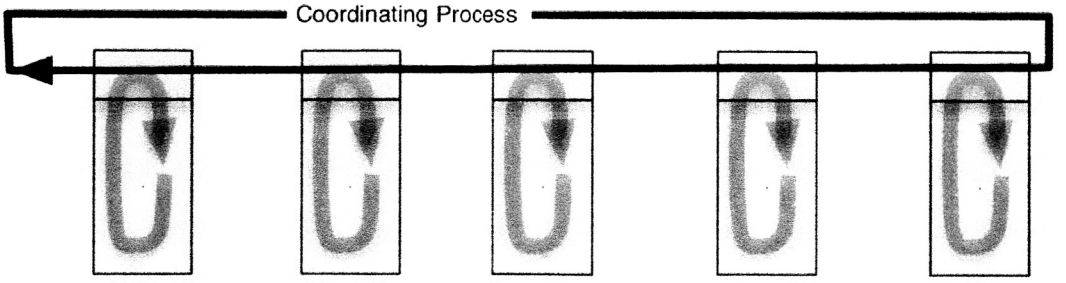


Figure 1: Flow of Information in the PIM Model

a PIM may support processes that execute purely locally on that platform, running continuously there. For example, low-level sensory processing that does not require intervention from high-level control is best done locally, and local execution controllers may manage local physical effectors. Such "reactive" processes may use up the bulk of the processing power on many components, and can run asynchronously with the coordinating process. Note that most data can be maintained locally on individual components, with the data accessible to the coordinating process only when it is resident. Note also that computation involving that data may still proceed as the process is running on another component as long as the necessary information is cached as part of the CP state and moves with the process. When the need arises to access data not locally resident or cached, the computation must wait until the coordinating process is again resident on the component where the data is stored.

3. Understanding the tradeoff between residency time and coordinated action

One of the major advantages of this model is that the programmer of the CP need not be overly concerned with the system-level details of how the CP moves between the components of a PIM. Nevertheless, these details will affect system performance in ways that require new modes of analysis. For example, the rate at which the CP moves amongst components must be fast compared to the required reactivity of the PIM.

We believe it is feasible in many important applications to cycle the CP amongst the components quickly enough that all critical coordination decisions for a component can become available (i.e., resident) in time to appropriately change its behavior. There are, however, interesting tradeoffs between computation and coordination in setting the length of time that the CP is resident on each component of a PIM. There are competing forces to balance:

A *longer* residency reduces the total fraction of time lost to transmission delays, thereby increasing the computational efficiency of the PIM at the cost of increasing the latency of the CP as it moves amongst the process, thereby decreasing the coordination and reactivity of the PIM. Conversely, a *shorter* residency time enhances the system's ability to coordinate overall responses to new and unexpected events since the overall cycle time of the CP will be shorter. But as we reduce the residency time, we increase the ratio of the overhead associated with moving the CP and thus decrease the computation available to the PIM for problem solving. In the extreme case, this could lead to a new form of thrashing, where little computation relevant to coordination is possible because all cycles are being used to move the CP from one component to the next.

Note that this tradeoff could be explicitly monitored and balanced during execution. For instance, a PIM could detect the approach of thrashing and take action to avoid it by, for example, increasing residency time. In another situation, when faced with the sudden need for increased coordination, it might temporarily decommission some components, thus decreasing the cycle time of the CP amongst the remaining components of the PIM without reducing the residency time.

A key requirement of our model is that the time taken to cycle the CP between components is small compared to the reaction time needed by the system. Conditions under which this assumption might fail include situations involving limited bandwidth between components (such as under water) or where remote communication fails altogether, but these conditions will pose

difficulties for any distributed system architecture. It will be important that designers of a PIM make wise decisions as they fit the design of the PIM to the environment in which it must operate. The proposed research will strive to better understand and characterize the trade-offs inherent in the PIM architecture.

4. Advantages of the PIM Architecture

We believe that the PIM architecture will have substantial advantages over the traditional models. A large part of this initial project would involve providing an initial analysis, and in some cases, experimental evidence, that these advantages can be attained. Here are some of the key intuitive advantages that need to be explored and validated.

Relative Simplicity of Code: Collections of agents or robots are notoriously difficult to manage because of the difficulties in maintaining a sufficiently coherent global state and the problems in distributed decision-making across several processes. The PIM model alleviates these difficulties and greatly simplifies the programmer's task.

Robustness: A key issue if how the system behaves in the face of losing components. If a component fails while the coordinating process is not resident, the operating system needs only to re-route the CP update around the missing component (using conventional network-management technology) and then the PIM will continue to operate with little effect (except that the cycle time will be reduced, thereby improving the responsiveness of the PIM!). If a component is destroyed while the CP is actually resident, the overall system can continue to function without significant disruption by activating a slightly out-of-date copy of the CP from another component of the PIM.

Simplification of coordination: As previously discussed, when a collection of robots is conceptualized as a team of independent agents, many complex issues arise concerning how best to communicate and coordinate the activities of the team. In fact, much of the communication in such systems can involve negotiation between the agents. All such considerations are rendered irrelevant by this architecture; they appear, if at all, only in the form of conventional issues of data management within a conventional program.

Energy Consumption: In addition to simplifying communication, the PIM architecture also should reduce the *amount* of communication needed, resulting in considerable energy savings, for communication can be an order of magnitude more expensive than computation in robotic applications. While robotic agents must communicate extensively to coordinate activity, our model eliminates all point-to-point communication, involves no negotiation protocols, and eliminates the need to move large volumes of data.

Effective management of Data: In a PIM architecture, the computational process goes to the data, rather than moving the data to the computation. This is a big win in any application using modern sensors, where the amount of data dwarfs the footprint of the CP state. But this is not just a bandwidth issue. As sensors get smarter, they need to coordinate better to jointly interpret observations. The PIM model should excel in simplifying the coordination requirements for large networks of smart sensors.

Data Security: Since most data remains distributed and does not need to be communicated, this model should be very effective for applications where the data is sensitive and cannot be released. Our model allows access to the data but does not require that the actual data be

transmitted off-site; and since all transmission of information between processors is within the state of the central process, security issues can be handled by one code stream.

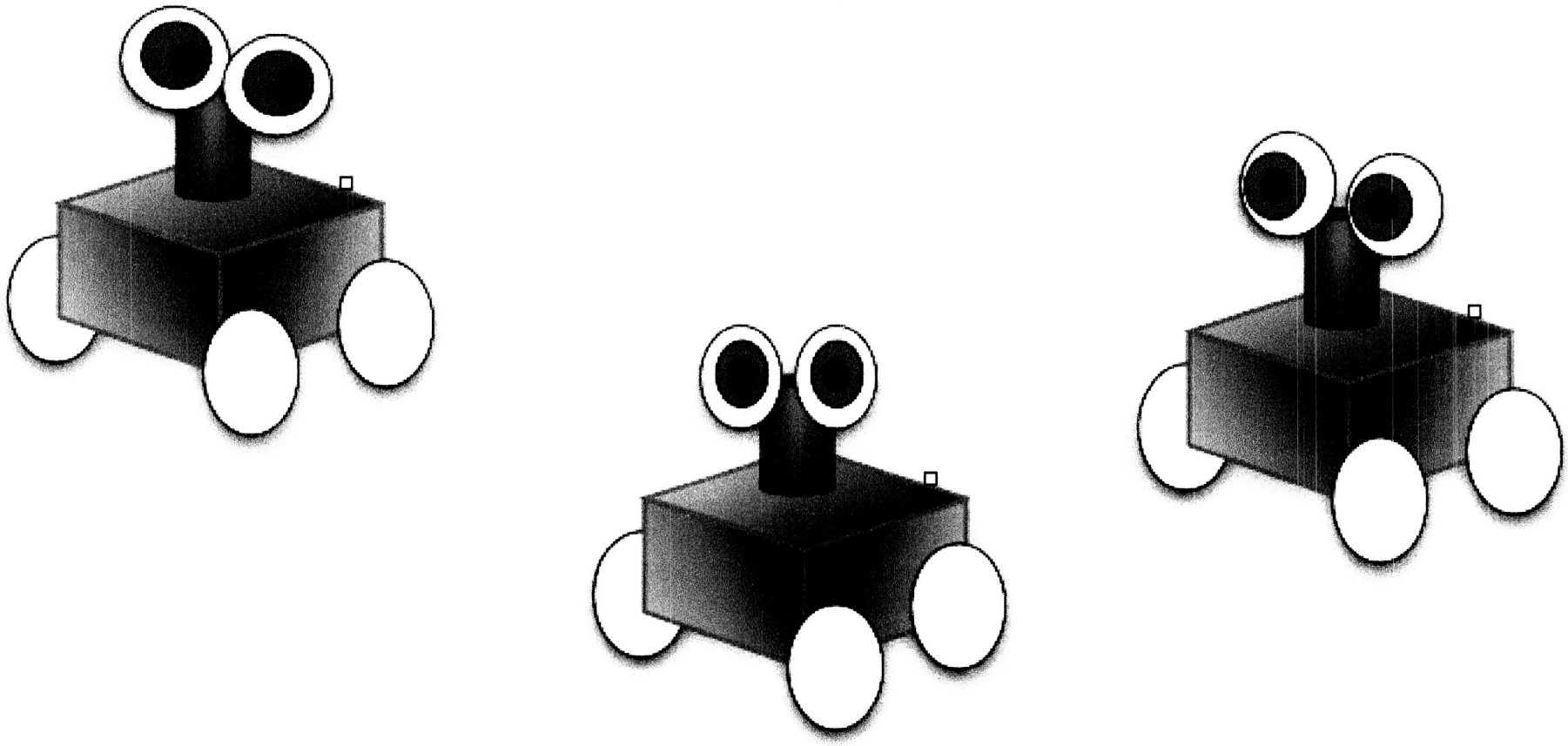
Effective Human/Machine Interfaces: The new architecture provides an effective range of options for *humans in the loop.* First and foremost, although the system is physically distributed, the human can interact with the system as an integrated whole (enhancing situational awareness). Furthermore, the architecture enables the human to be able to directly change, suspend or initiate coordinated action in essentially real-time (i.e., one cycle of the coordinating process). As mentioned above, if cycle time becomes insufficient to meet the coordination demands of the environment, it would also be simple for a human to decommission one or more components that are not critical to the current task.

One particularly general advantage of this architectural model is that it does not require elaborate protocols for communicating between agents, coordinating separate views of the situation or for achieving consensus before taking group action. Another is that the actual code of the CP can be largely written in a conventional manner appropriate for a single-processor platform, independently from the architectural complexities of the dynamics of the coordinating state. Taken together, these vastly simplify the top-level coding task, since the programmer should not have to think about *how* the processing is distributed among the components; and by allowing the use of conventional programming techniques, the overall system behavior is far more predictable than emergent behaviors of multi-agent systems. On the other hand, the idea of a single mechanism comprised of spatially separated parts that are independently mobile provides new opportunities for robotic planning, movement and force coordination and other applications. We expect that these will motivate new developments in programming techniques for advanced robotic control and "adaptive shape" robots.

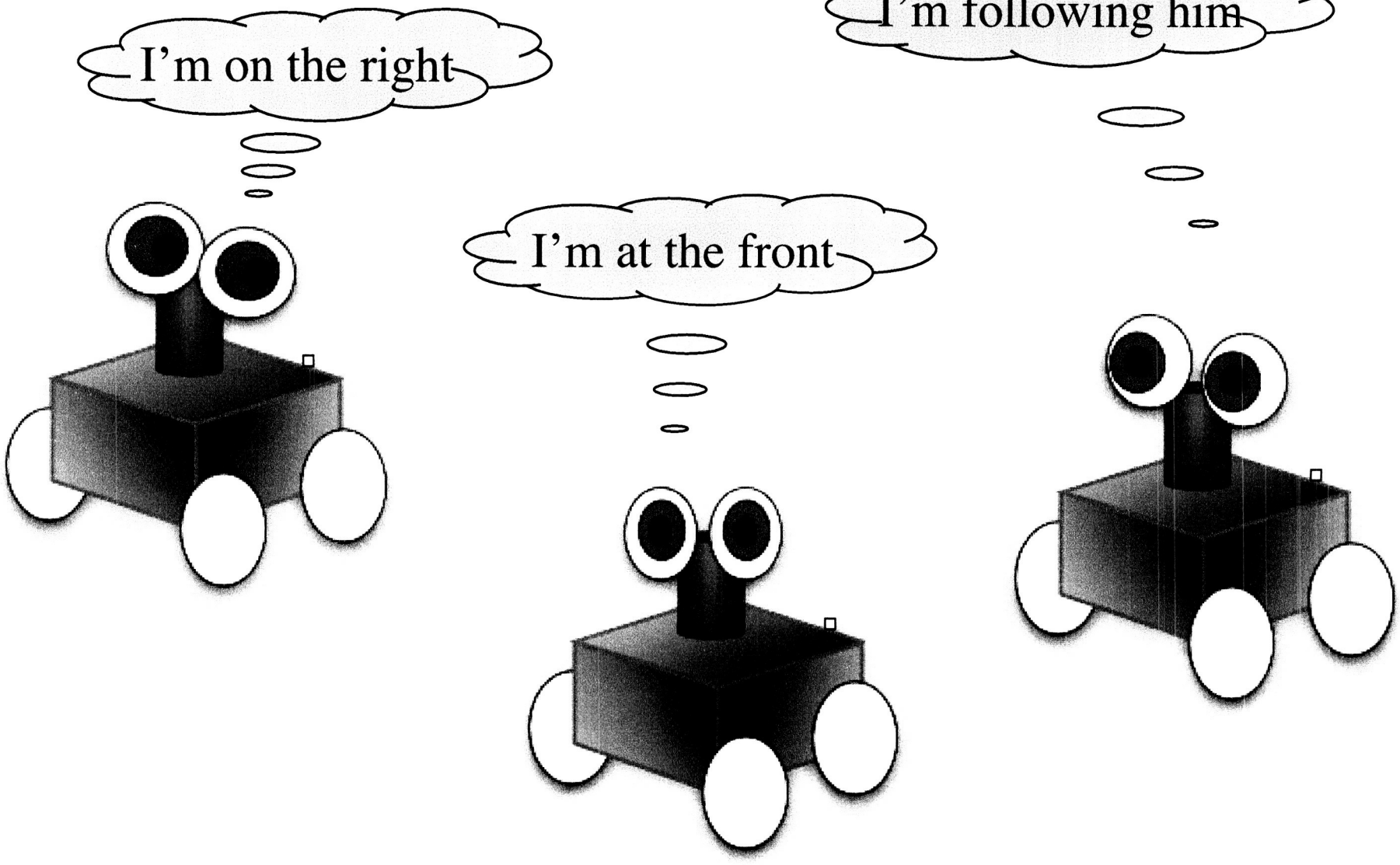
The Scatterbot

One Mind, Many Bodies

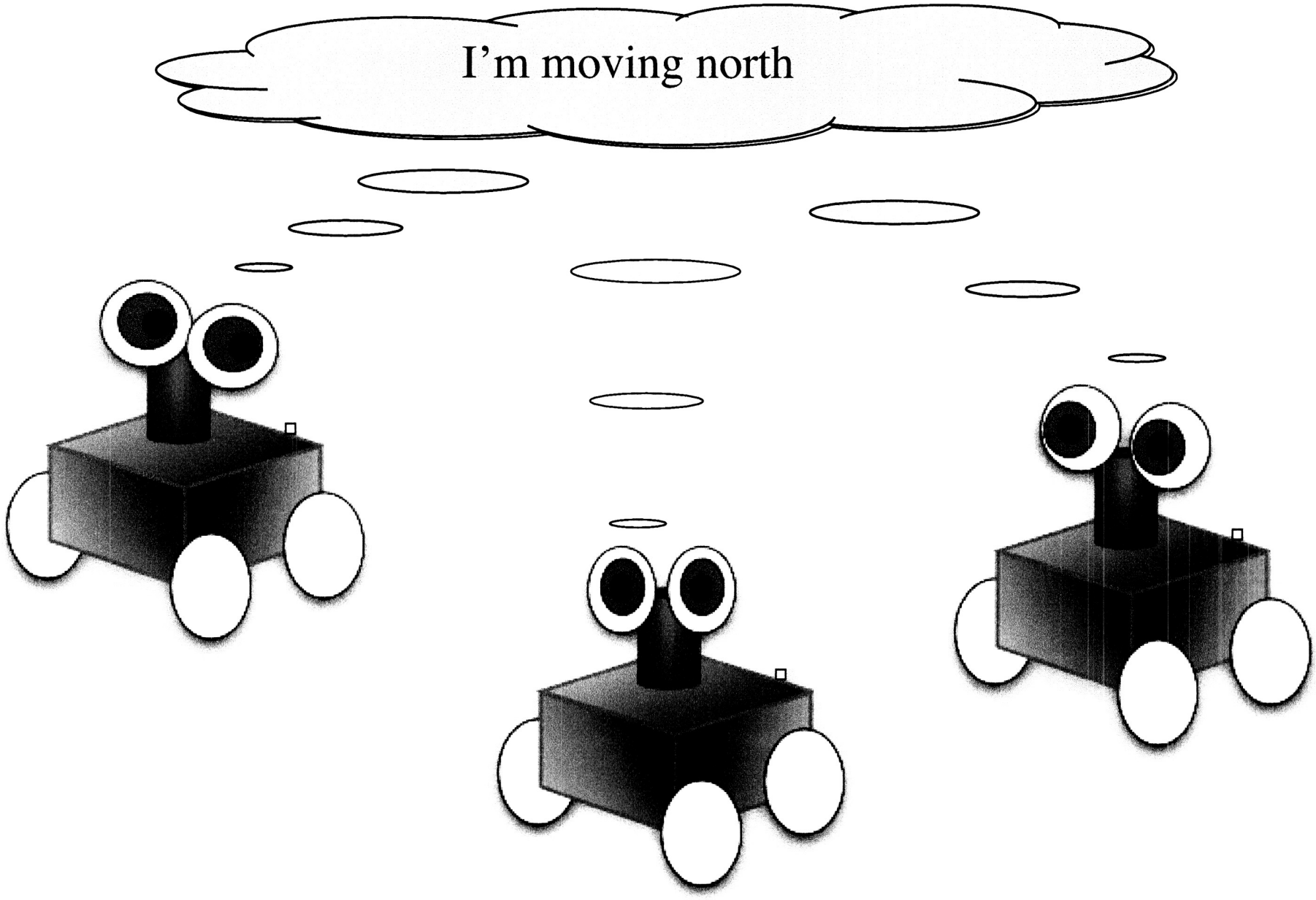
A crowd of little robots…

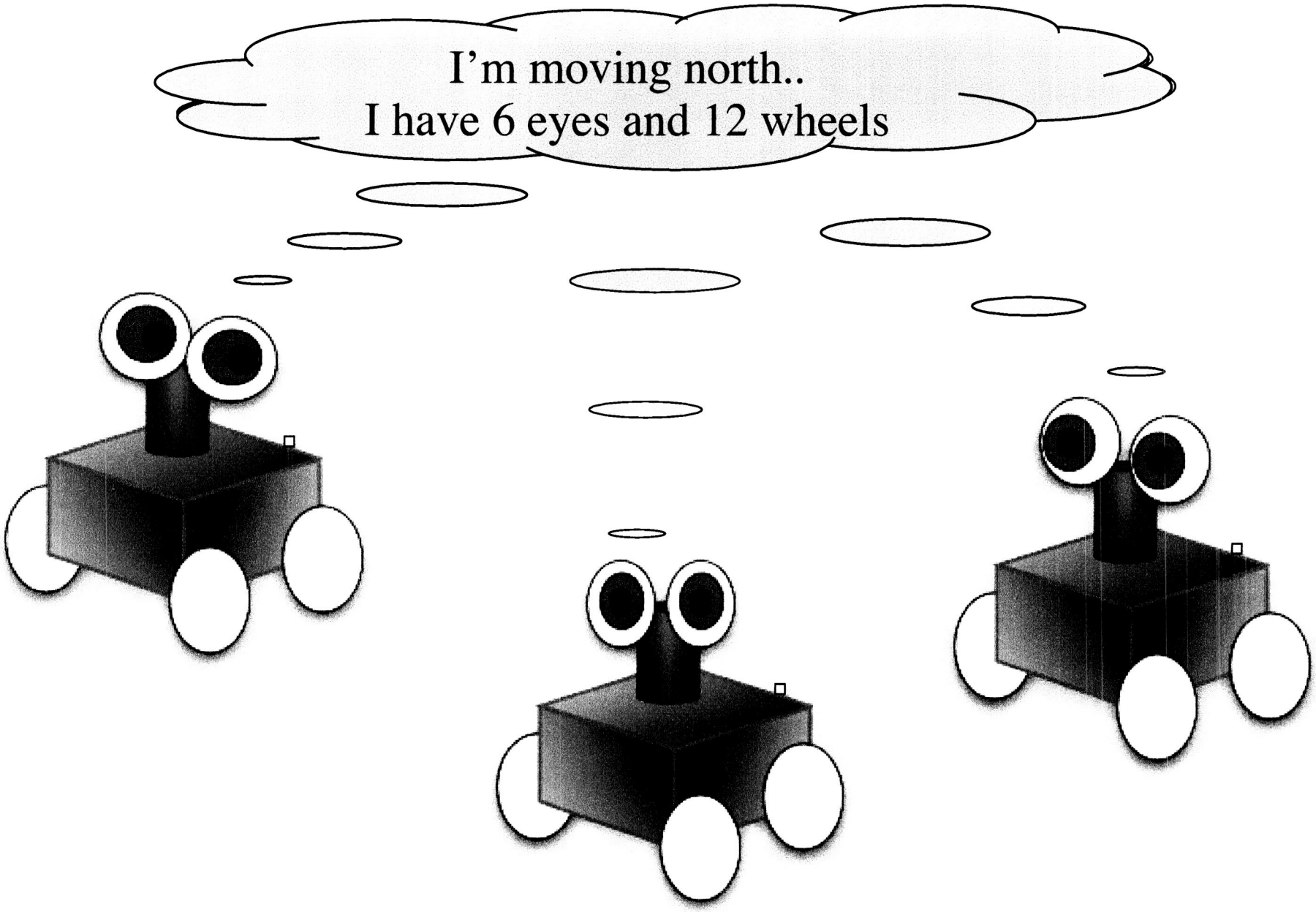


A *team* of little robots

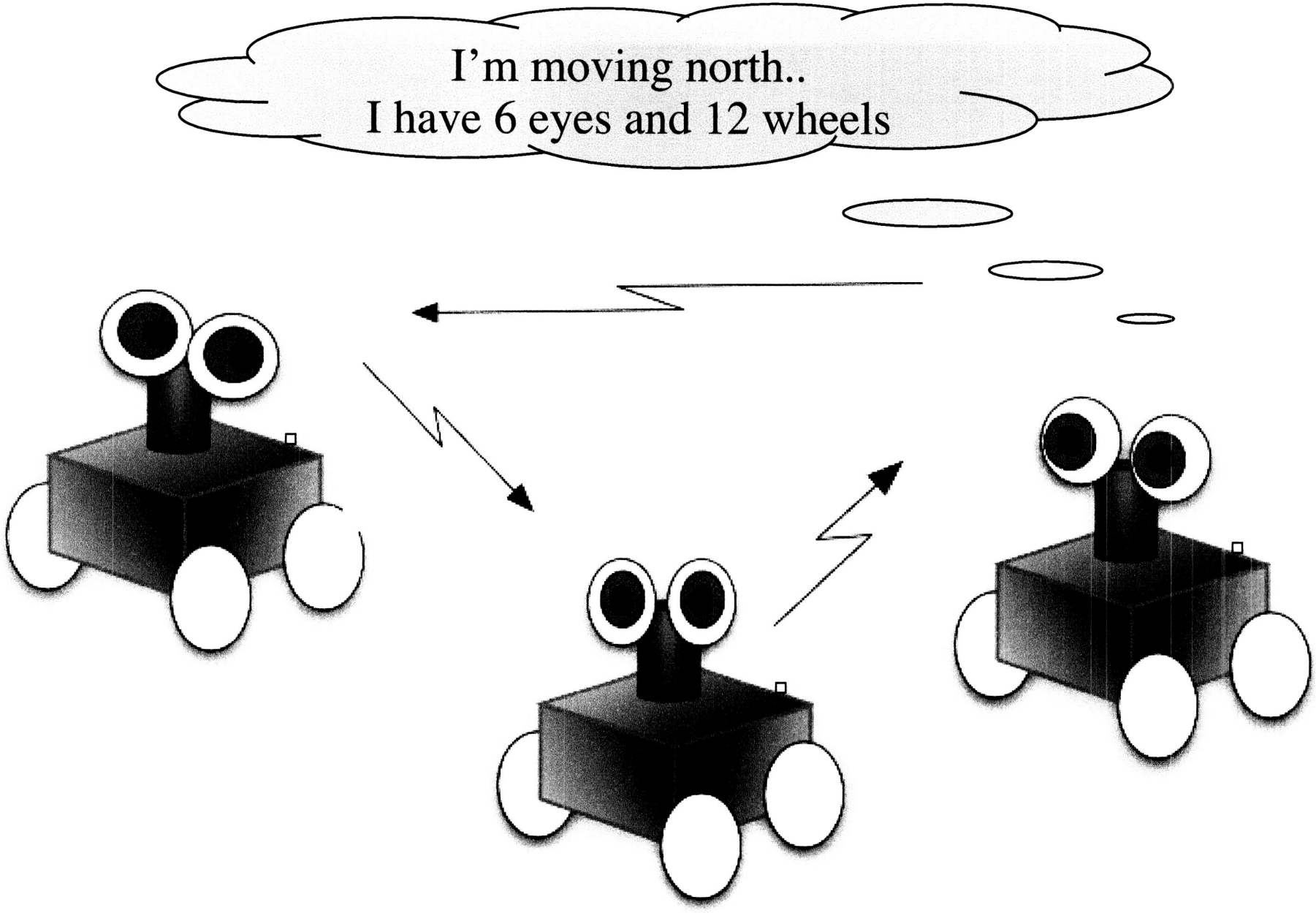


A *single* robot spread over a terrain

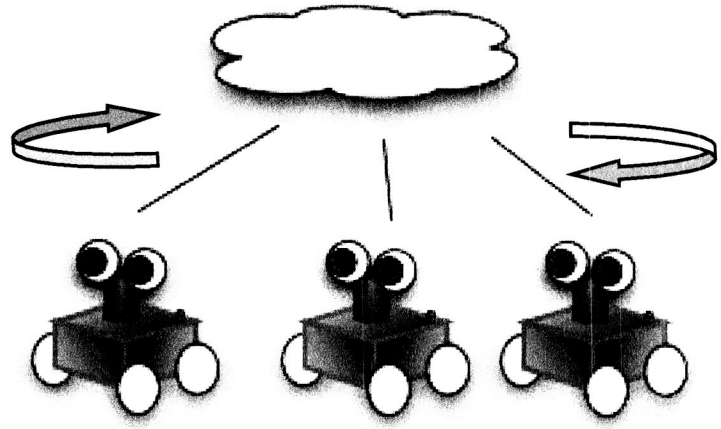



I'm moving north..
I have 6 eyes and 12 wheels

Crank up the timescale….

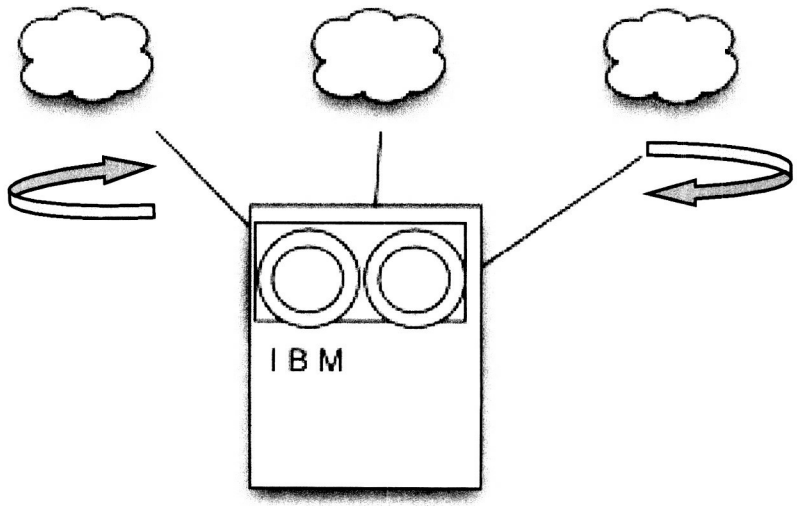


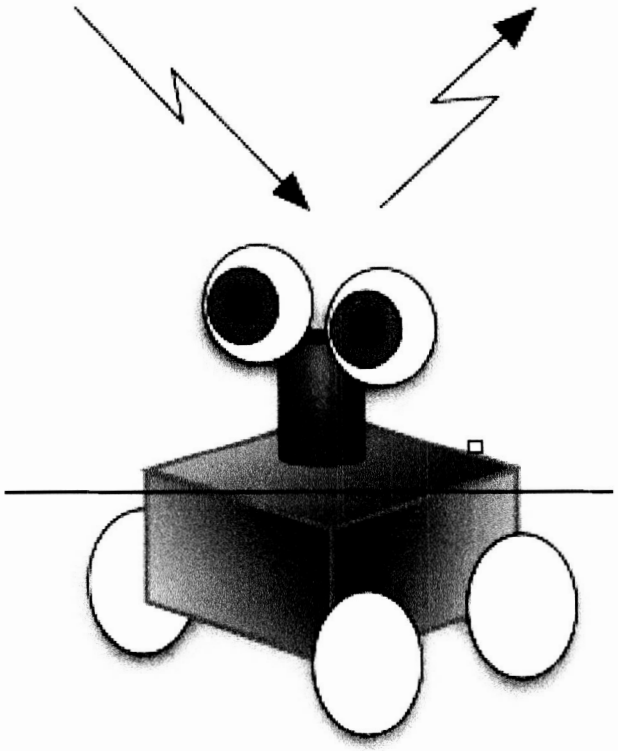
Timeborrowing =
One process shared between processors



inverse of

Timesharing =
One processor shared between processes





Architecture of a bot-part

Cognitive, mobile: Representation, planning, task model, etc.; common to all parts.

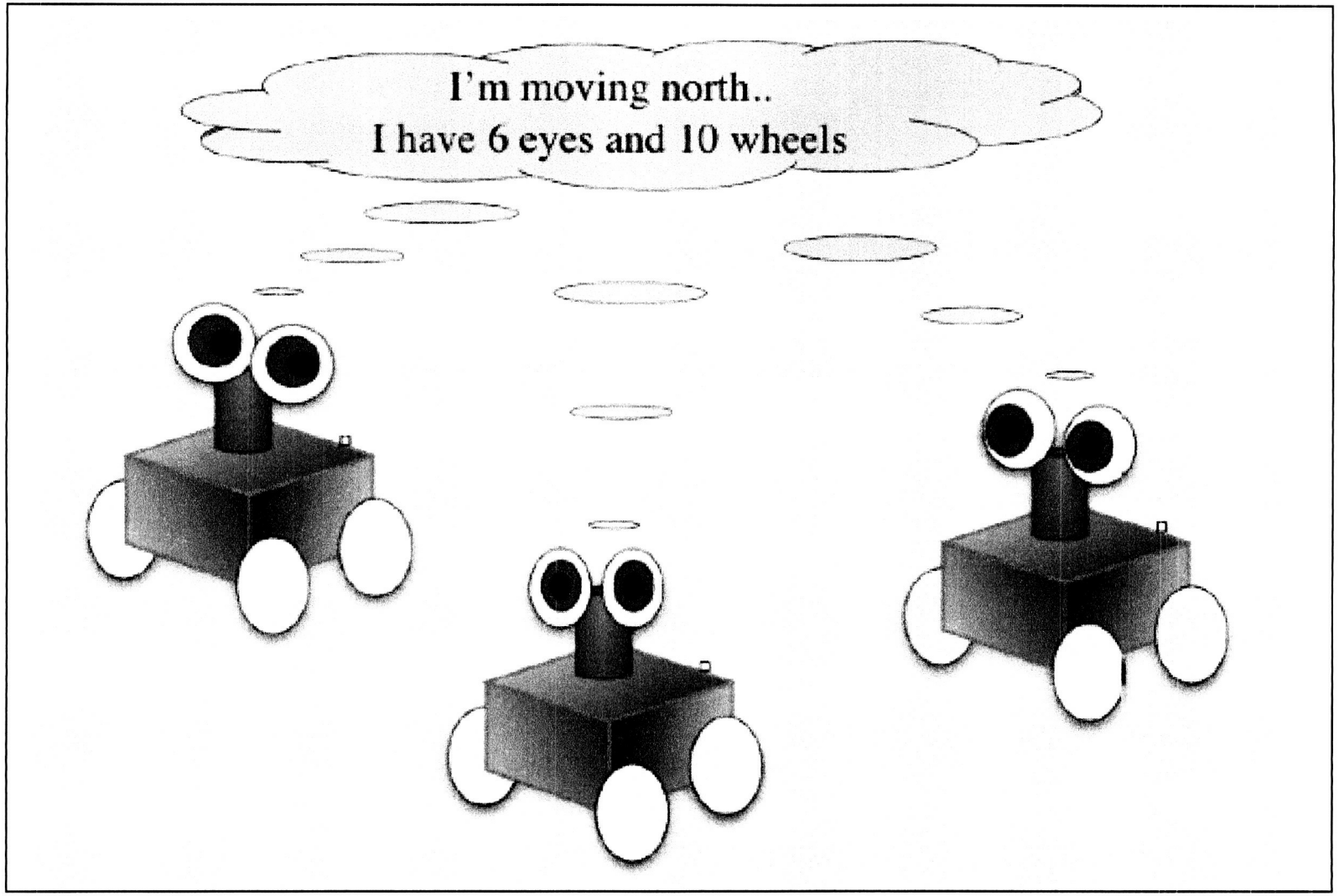
Timeborrowed by platform, integrates sensory information (from lizard level) and effector (motor) control (to lizard).

Lizard, fixed: Sensory processing & archiving, local mobility feedback, etc.; local to this botpart.

Provides computational and system support for cognitive level, maintains local activity, perhaps 'reflex' actions, network transmission & protocols.

e.g., round-robin: mobile mind runs for a while, moves to another bot-part and continues to run on this one. When a new version arrives…it replaces the current copy.

All the same, this is **one single entity** and is programmed on that basis.

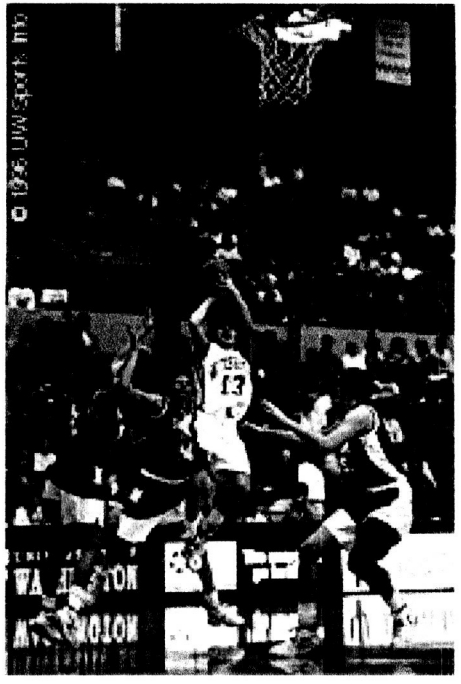


Scatterbots do not occur in nature; there are no biological examples. So *any* biological metaphor will be misleading to some extent.

Metaphor 1:

Telepathic Basketball

The whole team is
a *single organism* seeing
out of 10 eyes and
moving 5 bodies with
10 arms and legs.



Metaphor 2:

Plastic Man

Possible examples

1. Combat scatterbot; distributed weapon system.
2. Surveillance scatterbot
3. Rovers exploring terrain in opportunistic ways.
4. Spacecraft reconfiguring in flight
5. Manufacture, servicing: more flexible force coordinations
6. Convoys of robot vehicles on a highway

WHY BOTHER?

1. Robustness. *Space:* survives attrition; *Military*: impossible to kill; *Industry*: botparts can auto-replace.

2. Coordination. *Space*: acceleration, freefall require different configurations and rover botparts can configure for optimal mobility, or separate in emergency; *Military*: tactical behaviors; *Industry*: botparts can apply forces more directly, with greater sensitivity.

3. Opportunism. Scatterbots can loan, trade botparts and merge or divide.

4. Engineering. Conceptually clearer in architecture, allows richer spatial modeling, more complex actions.

WHY BOTHER?

1. Robustness. *Space:* survives attrition; *Military*: impossible to kill; *Industry*: botparts can auto-replace.

2. Coordination. *Space*: acceleration, freefall require different configurations and rover botparts can configure for optimal mobility, or separate in emergency; *Military*: tactical behaviors; *Industry*: botparts can apply forces more directly, with greater sensitivity.

3. Opportunism. Scatterbots can loan, trade botparts and merge or divide.

4. Engineering. Conceptually clearer in architecture, allows richer spatial modeling, more complex actions.

5. Fun!

OUR TASK, SHOULD WE DECIDE TO ACCEPT IT…

Identify **broad** constraints, principles, opportunities, risks, directions for future work.

Keep in the high blue sky, not get lost in details.

Don't fly too low.

Do not use the 'B' word.

Computational Mobility – An Overview

Niranjan Suri

ihmc

Institute for Human & Machine Cognition
http://www.ihmc.us/

Definition

- Movement of Data, Code, Computation, and Execution State from one System to Another Over a Network Link

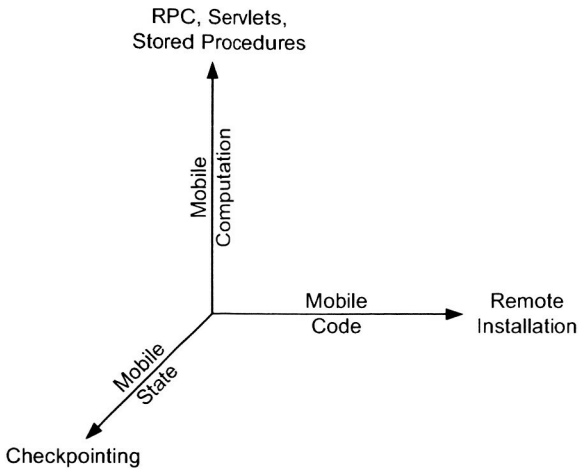
Types of Mobility

- Physical Mobility – Movement of Physical Objects in the Environment
- "Logical" Mobility – Movement of Bits Over a Communications Link from One Computer to Another
  - Types: Data, Code, Computation, Execution State
  - Mode: Push, Pull

Mobile Data

- Movement of Data From One Host to Another
- The Most Common Form of Mobility
  - Encompasses everything except code, computation, state
  - At some level – everything is data
- Not Important For Our Purposes

Mobility – Another Perspective



Mobile Code

- Allows executable code to be moved to a new host
- May use the push or pull model
  - Pull: Applets
  - Push: Remote Installation
- Code may be binary (intermediate or native) or source

Mobile Code

- Advantages:
  - Dynamically change capabilities
    - Download new code to add / change / update capabilities of platform
    - Remove code when no longer needed
- Problems:
  - Security concerns due to untrusted / unchecked code
    - Code could be malicious, buggy, and/or tampered

Mobile Computation

- Evolution of Remote Computation
  - RPC, RSH, RMI, Servlets, Stored Procedures, CORBA
- Allows one system to run a computation on another system
- Utilize resources on remote system
  - CPU, memory
- Access resources on remote system
  - Files, databases, etc.

Mobile State

- Evolution of State Capture
  - Checkpointing
- Allows execution state of a process to be captured and moved
- State may be machine specific or machine independent
- May contain
  - State of single or multiple threads
  - Code

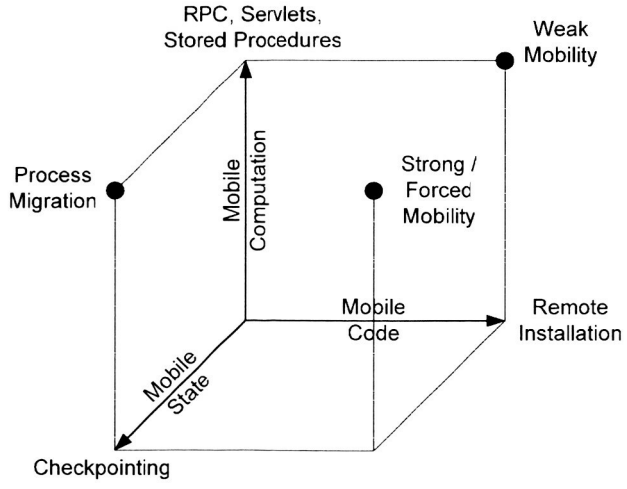
Mobility Matrix

	Data	**Code**	**Computation**	**Execution State**
Pull	Web Browsing, SETI@home	Applets, JavaScript, Jini	While You're Away (WYA)	While You're Away (WYA)
Push	FTP Upload	Remote Installation, Mobile Agents	RPC, RMI, Grid Computing	Process Migration

Weak -vs- Strong -vs- Forced

- Weak Mobility
  - Computing entity requests movement
  - Entity "restarts" execution after move operation
  - Combines Mobile Code and Mobile Computation
- Strong Mobility
  - Computing entity requests movement
  - Execution continues after movement
  - Combines Mobile Code, Mobile Computation, and Mobile State
- Forced Mobility
  - External, asynchronous request for movement
  - Execution continues after movement
  - Computing entity may not be aware of movement
  - Combines Mobile Code, Mobile Computation, and Mobile State

Weak -vs- Strong -vs- Forced



Weak Mobility Example One

```
public class Visitor
{
    public Visitor() {
        System.out.println ("Starting");
        move ("host1", this, "a");
    }
    public void a() {
        System.out.println ("On host one");
        move ("host2", this, "b");
    }
    public void b() {
        System.out.println ("On host two");
        move ("host3", this, "c");
    }
    public void c() {
        System.out.println ("On host three");
        move ("host1", this, "a");
    }
}
```

Weak Mobility Example Two

```
public class Visitor
{
    public Visitor() {
        System.out.println ("Starting");
        go ("host1", this);
    }
    public void run() {
        if (_where == 0) {
            System.out.println ("On host one");
            _where = 1;
            go ("host2", this);
        }
        else if (_where == 1) {
            System.out.println ("On host two");
            _where = 2;
            go ("host3", this);
        }
        else if (_where == 2) {
            System.out.println ("On host three");
            _where = 0;
            go ("host1", this);
        }
    }
    private int _where = 0;
}
```

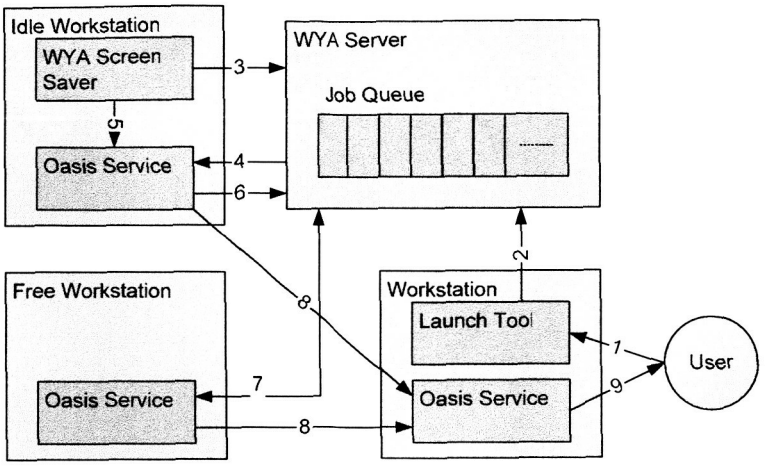
Strong Mobility Example

```
public class Visitor
{
    public Visitor()
    {
        System.out.println ("Starting");
        while (1) {
            go ("host1");
            System.out.println ("On host one");
            go ("host2");
            System.out.println ("On host two");
            go ("host3");
            System.out.println ("On host three");
        }
    }
}
```

Forced Mobility Example

- Visitor Not Appropriate
  - Mobility is dictated by external entity
- Examples:
  - Survivability
  - Load-balancing
- Concrete Example – While You're Away (WYA)
  - System for utilizing idle workstations
  - Abstraction – roaming computations

WYA Design



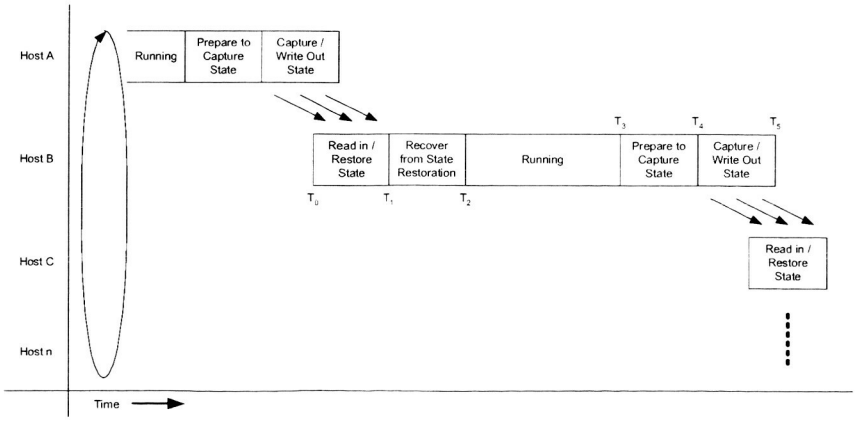
WYA Programming Abstraction

```
public class MyComputation extends RoamingComputation
{
    public void init (String args[])
    {
        // Perform any initialization required here
    }

    public void compute()
    {
        // Actual computations go here
    }

    public void reportResults()
    {
        // Report results back to the user here
    }
}
```

Forced Mobility Example Two

```
public class Jumper
{
    public Jumper() {
        System.out.println ("Starting");
        new Mover().start();
        while (1) {
            System.out.println ("hello, world");
        }
    }

    public class Mover extends Thread
    {
        public void run() {
            for (int i = 0; i < hosts.length; i++) {
                go (hosts[i]);
                Thread.sleep (100);
            }
        }
    }
}
```

Process Cycle



Mobility Abstraction

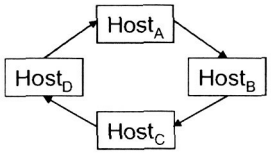
- Process is Continuously Moving
- Code Has no Knowledge of Current Host
- Code Prefixes Operation with a Scope that Identifies the Host
- Operation Gets Performed when Process is on that Host

Visitor Example Revisited

```
public class Visitor
{
    public Visitor()
    {
        System.out.println ("Starting");
        while (1) {
            h1.System.out.println ("On host one");
            h2.System.out.println ("On host two");
            h3.System.out.println ("On host three");
        }
    }
}
```

One Possible Realization…

- Hosts Form a Logical Ring
- Process is Created on one Host
- At Fixed Intervals (Timeslices?), Process is Migrated from $Host_n$ to $Host_{n+1}$
- Generic Operations May be Performed on Any Host
- Operations Qualified by a Host will be Performed only on that Host
  - Runtime system blocks until process is on required host
  - Runtime system possibly leaves process on required host until operation is completed
  - A form of critical section



Another Example

```
public class WasteTime
{
    public WasteTime()
    {
        System.out.println ("Starting");
        while (1) {
            float a = h1.readValue();
            float b = Math.sin (a);
            float c = Math.cos (b);
            h2.writeValue (c);
            float d = Math.acos (c);
            float e = Math.asin (d);
            h3.writeValue (e);
        }
    }
}
```

Variation on the Theme

- Process Migration Path is Determined by Operation to be Performed
  - If program wants to do something on $Host_p$, migrate directly to $Host_p$
- Could Result in Certain Hosts being Ignored
  - Undesirable if hosts deliver asynchronous events to process

Interesting "Performance" Questions

- What is a Good Timeslice?
- What is the Maximum Number of Hosts?
- When do you Start Thrashing?
- Answers Depend on Current State of the Art in Implementation
- What can we Project about the Future?

Interesting "Abstraction" Questions

- What is the Best Abstraction?
  - Is mobility dictated by the program?
  - Is program dictated by the mobility?
- What about Time?
  - Can `System.currentTimeMillis()` run anywhere?
  - Will cause clock synchronization problems
- Division Between Higher-order Functions and Lower-order Functions
- Splitting / Joining Groups
  - Equivalent of a fork() / join()?

Available Resource – Aroma VM

- Clean-room implementation
- State capture mechanism
- Dynamic, fine-grained resource control
  - Disk, Network, CPU
- JDK 1.2.2 compatible
  - Uses Java Platform API from JRE 1.2.2
  - No AWT / Swing
- Ported to Win32 (x86), Linux (x86), Solaris (SPARC)
- No Just-In-Time compilation (in progress)

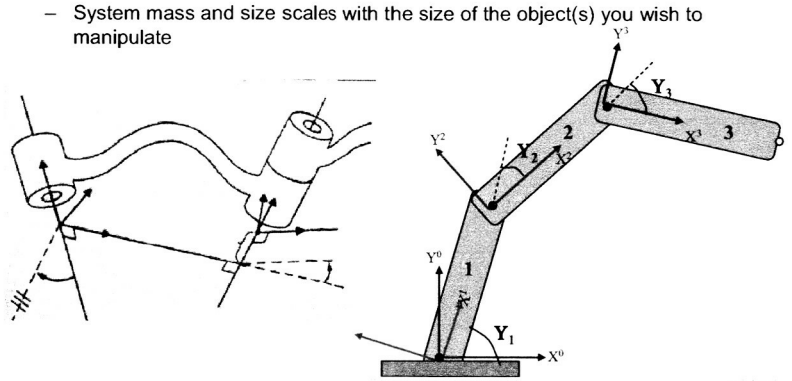
Actuation Mobility as a part of Computational Mobility

Butler Hine
NASA Ames Research Center

- **Multiple component nodes embody:**
  - Computational capability
  - Sensing capability
  - Actuation capability
- **Computational Mobility emphasizes:**
  - De-centralized processing and control
  - Robustness
  - Process adaptivity
- **But other modalities are also possible/desirable:**
  - Distributed sensing
    - Sensors residing in the component nodes are spatially distributed → improved coverage in space, time, and wavelength
  - Distributed Actuation
    - Actuation residing in the component nodes are also spatially distributed → force and torque manipulation beyond what is possible from a single node

Traditional Robotics

- **Kinematic chains**
  - Forces and torques are transmitted through *mechanical* linkages to the end effector
  - The system is limited in the external forces and torques it can exert through the end effector by the kinematic chain
  - System mass and size scales with the size of the object(s) you wish to manipulate



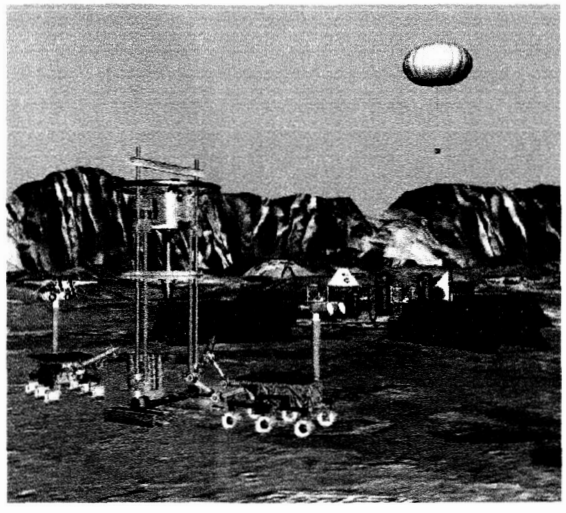
Computational Mobility Study

Distributed Actuation

- **Distributed Computational/Sensing/Actuation Nodes**
  - Forces and torques are transmitted by each unconnected node
  - External forces and torques are possible that are not limited by any mechanical connection
  - System mass becomes independent of the size of the object(s) you wish to manipulate

  - Possibilities:
    - Conformal Forces
      - Lifting/positioning large objects
      - Lifting/positioning delicate object
      - Multi-component assembly
      - Large size-scale compressive forces
      - Large size-scale expansion forces

Robot Team Scenario



Robotic exploration of Mars

- Mobile robots will serve as the remote sensors and data collectors for scientists.
- To create an outpost for such long-term exploration, robots need to
  - assemble solar power generation stations,
  - map sites and collect science data,
  - communicate with Earth on a regular basis.
- In one scenario, a large number of robots (20-30) are sent, many with different capabilities. Some of the robots specialize in heavy moving and lifting, some in science data collection, some in drilling and coring, and some in communication. The rovers have different, but overlapping, capabilities – different sensors, different resolutions and fields of view, even different mobility, such as both wheeled and aerial vehicles.

Robotic exploration of Mars

- Upon landing, the rovers search for a location suitable in size and terrain for a base station.
- Once such a location is found, rovers with appropriate capabilities form several teams to construct the base station capable of housing supplies and generating energy.
  - Two rovers carry parts, such as solar panels, that are too large for a single rover.
  - Complementary capabilities are exploited – for example, to align and fasten trusses, rovers with manipulators receive assistance from camera-bearing rovers that position themselves for advantageous viewing angles.

Robotic exploration of Mars

- Rover failures are addressed by dispatching a rover with diagnostic capabilities. The diagnostic rover can use its cameras to view the failed robot to see if it can be aided in the field (e.g., if it has a stuck wheel or is high-centered), or it may drag the rover back to the base station to be repaired by replacement of failed modules. In the meantime, another robot with the same (or similar) capabilities can be substituted, so as to complete the original task with minimal interruptions.

Robotic exploration of Mars

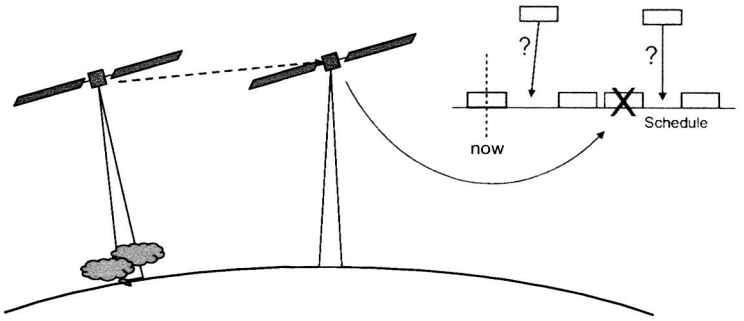
- At any given time, different teams of rovers may be involved in exploration, base-station construction/maintenance, and rover diagnosis/repair.
- Many tasks will be time critical, requiring execution within hard deadlines (e.g., repair of a failed power generation station) or synchronization with external events (communication satellite visibility, periods of sunlight).
- The teams form dynamically, depending on the task, environment, and capabilities and availability of the various robots to best meet mission requirements over time.
- The rovers negotiate their individual roles, ensure safety of the group and themselves, and coordinate their precise actions, attempting as a group to avoid unnecessary travel time, to minimize reconfiguration and wait time, and to prefer more reliable alternatives in cases of overlapping capabilities.
- The challenge is to keep all the robots healthy and busy in appropriate tasks, in order to maximize the scientific data collected.

Robotic exploration of Mars

- Similar scenarios exist for domains such as habitat construction, space solar power construction and maintenance, and Space Station maintenance.
  - For instance, consider an inspection robot that has identified a failed component on the Space Station. It tries to assemble a team of robots to replace the failed component. After negotiation, a courier robot (capable of retrieving the necessary replacement part) and a repair robot (capable of swapping-out the failed device) take responsibility for the repair task, leaving the inspection robot free to continue inspection. While the courier collects the replacement part, the repair robot evaluates the problem and plans its course of action, possibly seeking additional aid if it encounters unexpected difficulties it is unable to resolve. Upon arrival with the replacement part, the courier and repair robot tightly coordinate their actions, turning themselves into what is effectively a single high degree-of-freedom robot.

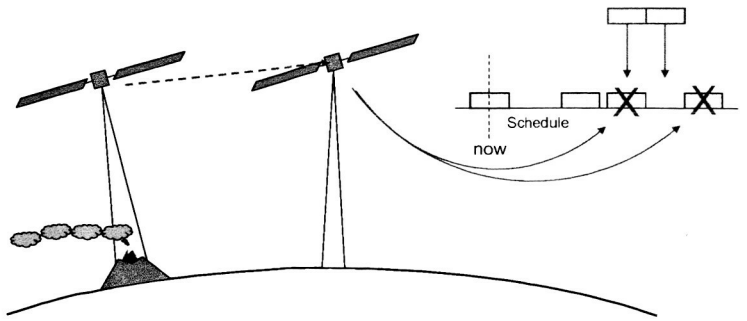
Coordinated Science Observation

Requires inter-satellite communication

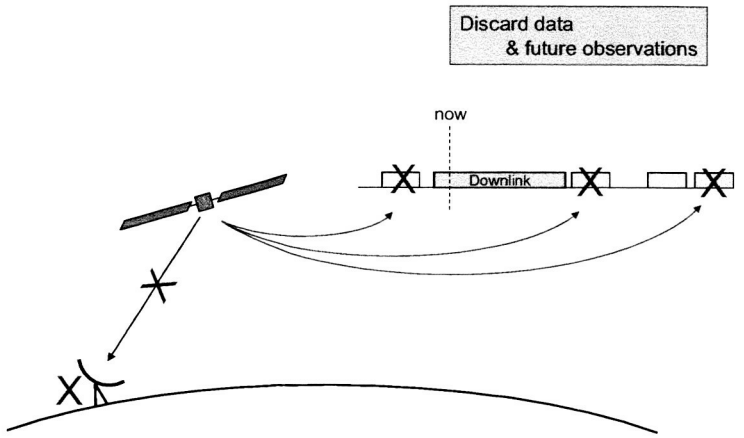


Coordinated Science Observation

Discard future observations
Insert new obs.



Coordinated Science Observation



A Procedural Language (Java) Abstraction for Programming the ScatterBot

Niranjan Suri

Goals

- Extend a standard Procedural Language – Java – to operate in a ScatterBot environment
- Hypothesize about how such a language might be realized
- Examine ScatterBot-specific issues that arise in the proposed language and implementation

Questions

- Is it right to call Java a procedural language?

Basics

- Assume that each bot part is represented by an object (an instance of some class)
  - The type of the object (i.e., the class) represents the type of the bot-part
  - We can leverage object-oriented notions of subclassing (is-a relationships) and containment (part-of relationships) to model bot-parts

"Types" of Objects

- There are Two Fundamental "Types" of Objects
  - Generic Java Objects (e.g., Strings, Vectors, etc.)
  - Objects "Bound" to Bot Parts (Bot Objects)
    - Similarity to Java native methods

Operations on Objects

- Three Basic Operations:
  - Read a variable
  - Write a variable
  - Call a method
- Same Operations on Bot Objects

Implementation Thoughts

- At the Java VM level, most bytecodes manipulate the operand stack and local variables - these can execute anywhere
- There are 3 types of bytecodes to worry about:
  - putstatic, getstatic
  - putfield, getfield
  - Invokevirtual, invokestatic, invokeinterface, invokespecial
- If any of these are executed on a Bot Object, the VM must execute the resultant operation only on the corresponding Bot part

Multiple Conditional Example

```
public class Test
{
    public void doSomething()
    {
        if (a && b && c && d && e) {
            System.out.println ("eureka");
        }
    }

    private boolean a;
    private boolean b;
    private boolean c;
    private boolean d;
    private boolean e;
}
```

Multiple Conditional Example

```
 0 aload_0
 1 getfield Test/a Z
 4 ifeq 43
 7 aload_0
 8 getfield Test/b Z
11 ifeq 43
14 aload_0
15 getfield Test/c Z
18 ifeq 43
21 aload_0
22 getfield Test/d Z
25 ifeq 43
28 aload_0
29 getfield Test/e Z
32 ifeq 43
35 getstatic java/lang/System/out Ljava/io/PrintStream;
38 ldc "eureka"
40 invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
43 return
```

Implementation Issues

- What About Multiple Threads?
- How is Synchronization Handled?
- Are Methods Blocking on Non-Blocking?
  - If you invoke a method to move a robot to a certain position, does the method return before or after the robot moves to that position?
- Concern: Does Allowing Multiple Threads Make the Program As Complex As A Multi-Agent System?

Potential Example To Elaborate

- Two robots, one stationary with a sensor package, one mobile with an arm to pick objects up
- The programming problem is to write code to make the mobile robot go pick things up and bring them back to the stationary robot to examine with the sensor package

Robot Coordination Example

```
// Explorer with two bot parts - a moving sample retriever and a stationary sample analyzer
public class Explorer
{
    private Vector _sampleSites;
    private Vector _sampleResults;
    private Analyzer _analyzer;
    private Retriever _retriever;

    public void run()
    {
        Enumeration e = _sampleSites.elements();
        while (e.hasMoreElements()) {
            // Move retriever to position
            Position p = (Position) e.nextElement();
            _retriever.moveTo (p);
            while (_retriever.moving()) {
            }
            // Pick up sample
            _retriever.pickupSample();

            // Move retriever to analyzer position
            _retriever.moveTo (X0, Y0);
            while (_retriever.moving()) {
            }
            // Analyze sample
            _sampleResults.addElement (_analyzer.analyzeSample());
        }
    }
}
```

Robot Coordination Example (2)

```
public class Analyzer
{
    public AnalysisResult analyzeSample (Sample s)
    {
        // Analyze sample and return result
        // Blocks while analysis is taking place
    }
}

public class Retriever
{
    public void moveTo (Position p)
    {
        // Start moving the retriever
        // returns immediately
    }

    public boolean moving()
    {
        // Return true if the retriever is still moving
    }
}
```

Enhancing the Coordination Example

- What is the model for multiple retrievers?
- Issues to consider:
  - Multiple retrievers need to be tasked in parallel
  - Retrievers may finish at different times
  - Retrievers may collide (or compete as they bring the samples to the analyzer)

Scatterbots

Daniel Cooke
Computer Science Department
Texas Tech University

Tuple Space

- Tuple Space is an abstraction for the communication path
- 'Bots are connected to the space

Center for Advanced Intelligent Systems

cAIs

Tuple Space

- Executive "lives" in the entire space – all bots and the TS
- 'Bots native codes live only on particular 'bot(s)
- Executive is only one who can place data in TS
- Natives can only get codes/data from the TS
- Executive can place data in the TS

Center for Advanced Intelligent Systems

cAIs

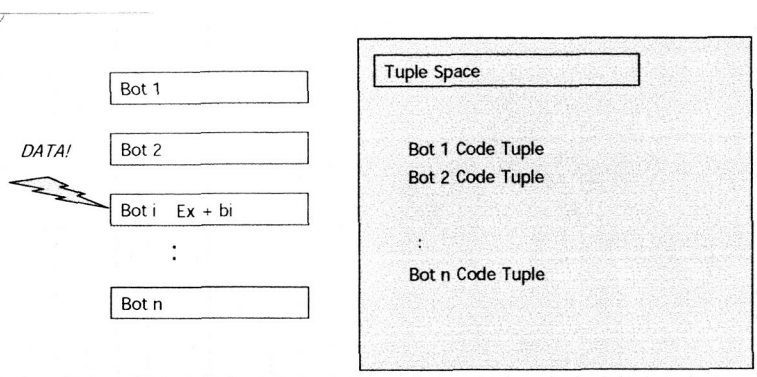
Tuple Space - Strict

- Executive Moves to Bot with Data to be Processed and then moves on with distilled data.
- Non-strict – the executive may place the distilled results in the tuple space and pair it with bots who need the information
- New bots can be added via the tuple space
- Bots can be removed via the tuple space

Center for Advanced Intelligent Systems

cAIs

Tuple Space - Strict

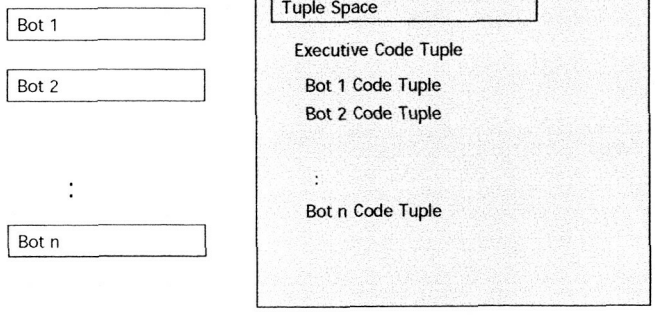


When a Bot(i) has important data the executive (Ex) moves with appropriate bot (bi) code to process data – minimizing movement of data

Center for Advanced Intelligent Systems

cAIs

Tuple Space - Strict

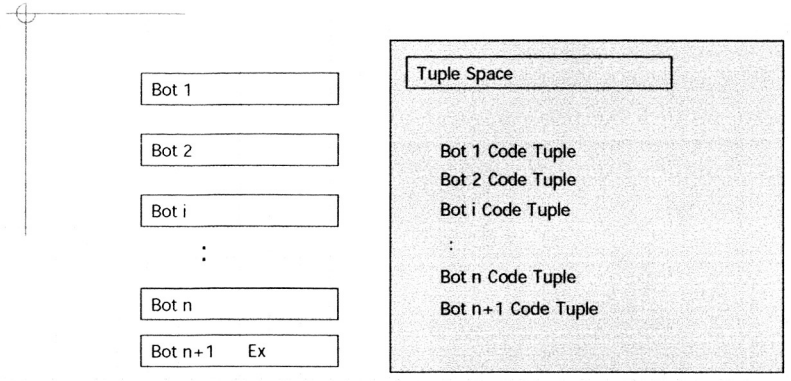


When a Bot(i) is no longer needed, he/she can inform the executive and remove code from TS

Center for Advanced Intelligent Systems

cAIs

Adding to theTuple Space - Strict



When a new bot (n+1) becomes available – he/she can add code to the tuple space and inform the executive

Center for Advanced Intelligent Systems